Compiler-Enhanced Incremental Checkpointing for OpenMP Applications

G. Bronevetsky, D. Marques, K. Pingali, S. McKee, R. Rugina

February 19, 2009

IEEE International Parallel & Distributed Processing Symposium
Rome, Italy
May 25, 2009 through May 29, 2009

Compiler-Enhanced Incremental Checkpointing for OpenMP Applications

Greg Bronevetsky
Lawrence Livermore National Lab
greg@bronevetsky.com

Daniel Marques
Ballista Securities
marques77@gmail.com

Keshav Pingali
The University of Texas at Austin
pingali@cs.utexas.edu

Sally McKee
Chalmers University of Technology
mckee@chalmers.se

Radu Rugina
VMWare
radu.rugina@gmail.com

***Abstract*—As modern supercomputing systems reach the peta-flop performance range, they grow in both size and complexity. This makes them increasingly vulnerable to failures from a variety of causes. Checkpointing is a popular technique for tolerating such failures, enabling applications to periodically save their state and restart computation after a failure. Although a variety of automated system-level checkpointing solutions are currently available to HPC users, manual application-level checkpointing remains more popular due to its superior performance. This paper improves performance of automated checkpointing via a compiler analysis for incremental checkpointing. This analysis, which works with both sequential and OpenMP applications, significantly reduces checkpoint sizes and enables asynchronous checkpointing.**

I. Introduction

Dramatic growth in supercomputing system capability from tera-flops to peta-flops has resulted in dramatically increased system complexity. Despite significant efforts to reduce the complexity of system software, the component complexity of these machines is still rising with rising node and core counts. Large-scale systems like IBM BlueGene, RoadRunner and Cray XT have grown to more than 100k processors and tens of TBs of RAM; upcoming designs promise to significantly exceed these limits. While these machines are made from high-quality components, their increasing size makes them increasingly vulnerable to faults, including hardware breakdowns [14] and soft errors [9].

Checkpointing is a common technique for tolerating failures. Application state is periodically saved to reliable storage, and is used to roll the application back to this prior state on failure. However, automated checkpointing can be very expensive due to the size of saved data and amount of time the application loses while blocked. For example, dumping all of RAM on a 128K-processor BlueGene/L supercomputer to a parallel file system takes approximately 20 minutes [12]. This cost can be reduced via incremental checkpointing [13], where application writes are tracked by a runtime monitor. If the monitor detects no modifications to some memory region between two adjacent checkpoints, that region is omitted from the second checkpoint, thereby reducing its size. Prior work has explored a variety of monitors, including virtual memory fault handlers [8], page table dirty bits, and cryptographic encoding techniques [4].

When application writes are tracked using virtual memory fault handlers, checkpointing can be optimized via "copy-on-write checkpointing" or, more generally, "asynchronous checkpointing". At each checkpoint, all pages to be checkpointed are marked non-writable and placed on a write-out queue. While the application continues executing a separate thread asynchronously saves pages on the write-out queue. When the checkpointing thread is finished saving a given page, the page is marked writable. If the application tries to write to a page that hasn't yet been saved, the segmentation fault handler is called, a copy of the page is placed in the write-out queue, and the application resumes execution. Asynchronous checkpointing thus decouples the application's execution from the checkpointing operation and allows checkpointing to be spread over a longer period of time. This reduces both the amount of time the application blocks on the checkpoint and the pressure on the I/O system and the network.

While prior work focuses on runtime techniques for monitoring application writes, this paper presents a compile-time analysis for tracking such writes. Given an application that has been manually annotated with calls to a `checkpoint` function, for each array the analysis identifies points in the code such that either:

- there exist no writes to the array between the point in the code and the next checkpoint and/or
- there exist no writes to the array between the last checkpoint and the point in the code

When the analysis can prove that some array is not modified along all paths between two checkpoints, this array is omitted from the second checkpoint. Furthermore, since each array can be asynchronously saved during the time period from the last pre-checkpoint write to the array until the first post-checkpoint write, the analysis supports asynchronous checkpointing. This exceeds the capabilities of runtime techniques because asynchronous checkpointing can begin even before a checkpoint. However, because it works at array granularity rather than the page- or word-granularity of runtime monitoring mechanisms, its decisions may be more conservative. Furthermore, the analysis assumes that when the application reaches a one potential checkpoint location, it can determine if it will take a checkpoint when it reaches the next location.

Prior work on compiler analyses for checkpoint optimization [10] [15] has focused on pure compiler solutions that reduce the amount of data checkpointed. Our work presents a hybrid compiler/runtime approach that uses the compiler to optimize certain portions of an otherwise runtime checkpointing solution. The resulting system thus both reduce the amount of data being checkpointed, and supports purely runtime techniques such as asynchronous checkpointing.

II. Compiler/Runtime Interface

Our incremental checkpointing system is divided into run-time and compile-time components. The runtime component checkpoints application memory inside `checkpoint` calls by either saving it in a blocking fashion or spawning a thread to checkpoint it asynchronously. Memory regions that do not contain arrays (a small portion of the code in most scientific applications) are saved in a blocking fashion. Arrays are saved in an incremental and possibly asynchronous fashion, as directed by the following annotations, which are placed by the compiler.

- `add_array(ptr, size)` Called when an array comes into scope to identify the array's memory region.
- `remove_array(ptr)` Called when an array leaves scope. Memory regions that have been added but not removed are treated incrementally by the checkpointing runtime.
- `start_chkpt(ptr)` Called to indicate that the array that contains the address `ptr` will not be written until the next checkpoint. The runtime may place this array on the write-out queue and begin to asynchronously checkpoint this array.
- `end_chkpt(ptr)` Called to indicate that the array that contains the address `ptr` is about to be written. The `end_chkpt` call must block until the checkpointing thread finishes saving the array. It is guaranteed that there exist no writes to the array between any checkpoint and the call to `end_chkpt`.

The runtime asynchronously checkpoints each array between the calls to `start_chkpt` and `end_chkpt` that refer to it. If `start_chkpt` is not called for a given array between two adjacent checkpoints, this array is omitted from the second checkpoint because it was not written to between the checkpoints.

Original Code	Transformed Code
`checkpoint()`	`checkpoint()`
`...`	`...`
`A[...]=...`	`end_chkpt(A)`
`...`	`A[...]=...`
`for(...) {`	`start_chkpt(A)`
`...`	`...`
`B[...]=...`	`for(...) {`
`...`	`...`
`}`	`end_chkpt(B)`
`...`	`B[...]=...`
`checkpoint()`	`...`
	`}`
	`start_chkpt(B)`
	`...`
	`checkpoint()`

Fig. 1. Transformation example

Figure 1 shows an example of how an application may be transformed to use this API. The original code contains two `checkpoint` calls, with assignments to arrays `A` and `B` in between. The code within the `...`'s contains no writes to `A` or `B`. The analysis transforms the code to include a call to `end_chkpt(B)` immediately before the write to `B` and a `start_chkpt(B)` call at the end of `B`'s write loop. This difference in placement is because a `start_chkpt(B)` call inside the loop may be followed by writes to `B` in subsequent iterations. Placing the call immediately after the loop ensures that this cannot happen.

III. Compiler Analysis

The incremental checkpointing analysis is a dataflow analysis that consists of forward and backward components. The forward component, called the *Dirty Analysis*, identifies the first write to each array after a checkpoint. The backward component, called the *Will-Write* analysis, identifies the last write to each array before a checkpoint.

A. Basic Analysis

For each array at each node n in a function's control-flow graph(CFG) the analysis maintains two bits of information:

- $mustDirty[n](array)$: $True$ if there *must* exist a write to $array$ along *every* path from a `checkpoint` call to this point in the code; $False$ otherwise. Corresponds to the dataflow information immediately *before* n.
- $mayWillWrite[n](array)$: $True$ if there *may* exist a write to $array$ along *some* path from a this point in the code to a `checkpoint` call; $False$ otherwise. Corresponds to the dataflow information immediately *after* n.

$$
\begin{aligned}
&mustDirty[n](array) = \\
&\quad \text{if } (n = \text{first node}) False \\
&\quad \text{else } \textstyle\bigcap_{m \in pred(n)} mustDirtyAft[m](array)
\end{aligned}
$$

$$
\begin{aligned}
&mustDirtyAft[m](array) = \\
&\qquad [\![m]\!](mustDirty[m](array), array)
\end{aligned}
$$

$$
\begin{aligned}
&mayWillWrite[n](array) = \\
&\quad (\text{if } n = \text{last node}) False \\
&\quad \text{else } \textstyle\bigcup_{m \in succ(n)} mayWillWriteBef[m](array)
\end{aligned}
$$

$$
\begin{aligned}
&mayWillWriteBef[m](array) = \\
&\qquad [\![m]\!](mayWillWrite[m](array), array)
\end{aligned}
$$

Statement m	$[\![m]\!](val, array)$
`array[expr] = expr`	$True$
`checkpoint()`	$False$
other	val

Fig. 2. Dataflow formulas for Dirty and Will-Write analyses

Figure 2 presents the dataflow formulas used to propagate this information through the CFG. The Dirty and Will-Write analyses start at the top and bottom of each function's CFG, respectively, in a state where all arrays are considered to be clean (e.g., consistent with the previous and next checkpoint, respectively), meaning that their write bit is $False$. As they propagate forward and backward through the CFG, respectively, at each array write they set the corresponding array's bit to $True$. When each analysis reaches a `checkpoint` call, it resets the state of all arrays to $False$. For the Dirty Analysis this is because all arrays are checkpointed, which makes them clean. For the WillWrite Analysis, at the point immediately before a checkpoint there exist no writes to any arrays until the next checkpoint, which is the immediately following statement.

Figure 3 shows the algorithm used to annotate the application source code with calls to `start_chkpt` and`end_chkpt`. The algorithm adds such calls in three situations. First, it inserts `end_chkpt(array)` immediately before each write to $array$ for which there exists *some* path that starts from a call to `checkpoint`

```
foreach (array $array$), foreach (CFG node $n$)
  // if $n$ is the first write to $array$
  // since the last checkpoint call
  if($mustDirty[n](array) = False \wedge$
      $mustDirtyAft[n](array) = True$)
    place end_chkpt($array$)
    immediately before $n$
  // if $n$ is the last write to $array$ until the next
  // checkpoint call
  if($mayWillWriteBef[n](array) = True \wedge$
      $mayWillWrite[n](array) = False$)
    place start_chkpt($array$)
    immediately after $n$
  // if $n$ follows the last write on a branch where
  // $array$ is no longer written
  if($mayWillWriteBef[n](array) = False \wedge$
    ($\exists m \in pred(n).$
      $mayWillWrite[m](array) = True$))
      place start_chkpt($array$)
      on edge $m \rightarrow n$
```

Fig. 3. Transformation for `start_chkpt`/`end_chkpt` insertion

and contains no writes to $array$. Second, it inserts `start_chkpt(array)` immediately after each write to $array$ that is not followed by any write to $array$ along *any* path that leads to a `checkpoint` call. Third, it inserts `start_chkpt(array)` on each CFG branching edge $m \rightarrow n$ if $mayWillWrite[n](array)$ is $True$ at m, but $False$ at n as a result of dataflow information being merged at branching point m. This situation occurs commonly in loops that write to an array. In this case, $mayWillWrite[n](array)$ is $True$ at all points in the loop body, since $array$ may be written in subsequent loop iterations. The flag becomes $False$ on the edge that branches out of the loop, and the compiler inserts the `start_chkpt(array)` call on this edge.

Because the Dirty analysis is based on *must-write* information, `end_chkpt` calls are conservatively placed as late as possible after checkpoints. Furthermore, the Will-Write analysis' use of *may-write* information conservatively places `start_saves` as early as possible before checkpoints.

Figure 4 provides an example of how the analysis operates on the code from Figure 1, showing the $mustDirty$ and $mayWillWrite$ states at each CFG node as well as the final transformed code. Observe that the `start_chkpt` and `end_chkpt` calls are inserted at points in the CFG where an array's $mustDirty$ and $mayWillWrite$ state, respectively, changes from $False$ to $True$.

Original Code	Code with Dirty States	Code with Will-Write States	Transformed Code
`checkpoint();`	`checkpoint(); [A→F,B→F]`	`checkpoint(); [A→T,B→T]`	`checkpoint();`
`...`	`... [A→F,B→F]`	`... [A→T,B→T]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`A[...]=...; [A→F,B→T]`	`end_chkpt(A);`
`...`	`... [A→T,B→F]`	`[A→F,B→T]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→F]`	`for(...) { [A→F,B→T]`	`start_chkpt(A);`
`...`	`... [A→T,B→F]`	`... [A→F,B→T]`	`...`
`B[...]=...;`	`B[...]=...; [A→T,B→F]`	`B[...]=...; [A→F,B→T]`	`for(...) {`
`...`	`... [A→T,B→T]`	`... [A→F,B→T]`	`...`
`}`	`} [A→T,B→T]`	`} [A→F,B→T]`	`end_chkpt(B);`
`...`	`... [A→T,B→F]`	`... [A→F,B→F]`	`B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→F]`	`checkpoint(); [A→F,B→F]`	`...`
			`}`
			`start_chkpt(B);`
			`...`
			`checkpoint();`

Fig. 4. Analysis example

B. Loop-Sensitive Analysis

Although simple and effective, the above transformations can introduce high overheads into application loops. This can be seen in the transformed code in Figure 4. `start_chkpt(B)` is placed immediately after the loop that writes `B`, while `end_chkpt(B)` immediately before the write to `B` itself. The reason for this difference is that `end_chkpt` is placed using *must-write* information, and the placement of `start_chkpt` uses *may-write* information. As a result, `end_chkpt(B)` is executed during each loop iteration, resulting in high overhead for small, deeply-nested loops, which are very common in scientific computing.

We address problem by augmenting the above analysis with a loop-detection heuristic, shown in Figure 5. This heuristic extends must-Dirty and may-WillWrite information used in Section III with may-Dirty information, which allows it to identify the pattern of dataflow facts that must hold at the top of the first loop that writes to an array after a checkpoint. Figure 5 shows the CFG of such a loop and identifies edges in the CFG where the various dataflow facts are $True$.

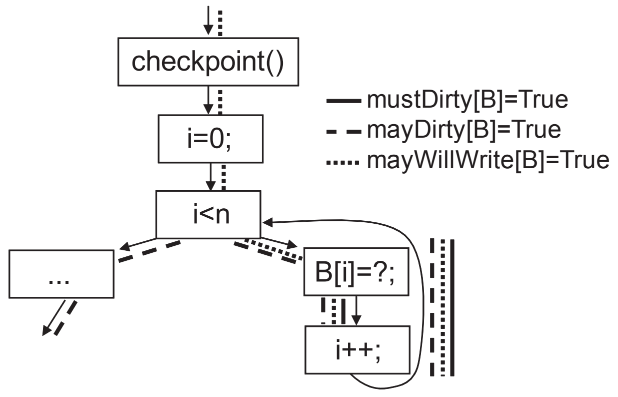


Fig. 5. Dataflow pattern for writes inside loops

The pattern at node $i < n$ is:

- $mustDirty[i < n](B) = False$
- $mayDirty[i < n](B) = True$
- $mayWillWrite[i < n](B) = True$
- $pred(i < n) > 1$

Furthermore, $mustDirty[p](B) = False$ at node p, the immediate predecessor of the loop test node $i < n$. Thus, placing `end_chkpt(B)` on the edge $p \rightarrow [i < n]$ ensures that `end_chkpt(B)` is called before any write to `B` and is not executed in every iteration of the loop.

Since this heuristic only applies to loops, it does not place `end_chkpt(A)` before the write to `A` in Figure 1. Thus, both rules are needed to ensure that `end_chkpt` is placed conservatively. We first use the loop-placement rule to place `end_chkpt(array)` before loops and then use the basic rule from Section III to place `end_chkpt(array)` in code locations that remain uncovered. An additional *EndChkpt-Placed* analysis ensures that we do not place `end_chkpt(array)` at nodes where there already exists an `end_chkpt(array)` on every path from any `checkpoint` call to the node. *EndChkpt-Placed* is a forward analysis that is executed as a separate pass from the Dirty and Will-Write passes. It maintains a bit of information for every array at every CFG node. $mustEndChkptPlaced[n](array)$ is set to $True$ if `end_chkpt(array)` is to be placed immediately before node n and set to $False$ if `start_chkpt(array)` is to be inserted at n. The latter rule ensures that the "exclusion-zone" of a given `end_chkpt(array)` call doesn't last past the next `checkpoint` call.

The loop-sensitive analysis implements this rule by maintaining for each CFG node n the following additional dataflow information:

- $mayDirty[n](array)$: $True$ if there *may* exist a write to $array$ along *some* path from a

`checkpoint` call to this point in the code; $False$ otherwise. Corresponds to the dataflow information immediately *before* n.
- $mustEndChkptPlaced[n](array)$: $True$ if *all* paths from any `checkpoint` call to this point in the code contain a point where a `end_chkpt(array)` call will be placed.

Figure 6 shows how this information is computed and Figure 7 provides he modified rules for placing `end_chkpt` calls. Figure III-B then extends the example in Figure 1 with $mustEndChkptPlaced$ information and shows how `end_chkpt` calls are placed.

```
foreach ($array$), foreach (CFG node $n$) in app
    if $placeEndChkptNode(n, array)$
        place end_chkpt($array$)
        immediately before $n$
    if $\exists m \in pred(n).$
            $placeEndChkptEdge(m, n, array)$
        place end_chkpt(array) on
        edge $m \rightarrow n$
```

Fig. 7. Loop-sensitive transformation for `end_chkpt` insertion

C. Inter-Procedural Analysis

We extend the above analysis with a context-insensitive, flow-sensitive inter-procedural analysis that applies the data-flow analysis from Section III-B to the CFG that contains all of the application's functions. When the analysis reaches a function call node for the first time, it computes a summary for that function by applying the dataflow analysis using the formulas in Figure 6, but with a modified lattice.

In addition to the standard $True$ and $False$, we extended lattice contains an additional $Unset$ state that appears below $True$ and $False$. When analyzing a given function all the dataflow facts for all arrays are initialized to $Unset$ at the start or end of the function (start for the forward analyses and end for the backward analysis). We then execute the standard analysis on the function using the extended lattice with $Unset$ treated as $False$ by the EndChkpt-Placed analysis. Array states that are not modified by a given pass remains $Unset$ at the end of a the pass. For the Dirty and Will-Write analyses this means that the array is not written to inside the function. For the EndChkpt-Placed analysis, this means that no `end_chkpt` calls are placed for this array inside the function. The function summary is then the set dataflow facts for all arrays at the opposite end of the function: end for the forward analyses and start for the backward analysis. The effects of a function call on dataflow state is computed by applying the function summary as a mask on all dataflow state. If $dataFlow[array] = Unset$ in the function summary, $array$'s mapping is not changed in the caller. Otherwise, it is set to $dataFlow[array]$.

IV. OpenMP Support

The increasing popularity of shared memory platforms for HPC (ranging from clusters of symmetric multiprocessors to large shared-memory machines like the SGI Altix) has led to the increased importance of the shared memory programming model. OpenMP is one of the most popular shared-memory APIs, with many applications written in either pure OpenMP or a combination of OpenMP and MPI. We have extended the above analysis to support multi-threaded OpenMP applications. OpenMP offers a structured fork-join parallelism model, with parallel regions of code identified using `#pragma omp parallel`. It also offers support for variable privatization, work-sharing, and synchronization. In light of prior work on shared memory checkpointing, our work has focused on *blocking* checkpointing tools such as C^3[5] [6] and BLCR[7], for these have proved to be the most portable. In particular, the analysis assumes that `checkpoint` calls are (i) global barriers across all threads, and (ii) every thread will execute the *same* `checkpoint` call as part of the same checkpoint. This is similar to OpenMP's semantics for placing `#pragma omp barrier`.

The intuition behind our analysis extension is that each thread is treated as a sequential application. The sequential analysis is applied to the application, ignoring any interactions among threads. This ensures that `start_chkpt` and `end_chkpt` calls are placed such that no thread:

- writes to `array` between a `start_chkpt(array)` call and a `checkpoint` call, or
- writes to `array` between a `checkpoint` call and an `end_chkpt(array)` call.

This alone is sufficient for code outside of `#pragma omp parallel` constructs and code that deals with private variables. This is because in both cases the array access patterns are sequential. However, it presents problems for parallel code that deals with shared variables. Consider the case of `start_chkpt(array)`, where `array` is shared. Although each thread is guaranteed to call `start_chkpt(array)` after the last pre-checkpoint write to `array`, the fact that one thread has called `start_chkpt(array)` does not mean that all threads are finished writing to array. As such, in the multi-threaded setting the checkpointing runtime is not allowed to begin asynchronously checkpointing `array` until *all* threads have called `start_chkpt(array)`. Similarly, the checkpointing runtime must finish checkpointing `array` when *any one* thread calls `end_chkpt(array)`.

$$mayDirty[n](array) = \begin{cases} False & \text{if } n \text{ = first node} \\ \bigcup_{m \in pred(n)} mayDirtyAfter[m](array) & \text{otherwise} \end{cases}$$

$$mayDirtyAfter[m](array) = [\![m]\!](mayDirty[m](array), array)$$

```
$mustEndChkptPlaced[n](array) =$
        $= \begin{cases} False & \text{if } n \text{ = first node} \\ \bigcap_{m \in pred(n)} mustEndChkptPlacedAfter[m](array) & \text{otherwise} \end{cases}$
$mustEndChkptPlacedAfter[m](array) =$
    if $\neg\ placeStartChkptNode(m, array)\ \wedge\ \neg\ \exists l \in pred(m).\ placeStartChkptEdge(l, m, array)$ then
        $False$
    else if $(placeEndChkptNode(m, array)\ \vee\ \exists l \in pred(m).\ placeEndChkptEdge(l, m, array))$ then
        $True$
    else $mustEndChkptPlaced[m](array)$

// end_chkpt(array) will be placed immediately before node $n$ if
$placeEndChkptNode(n, array) =$
    // node $n$ is the first write to $array$ since the last checkpoint
    $(mustDirty[n](array) = False \wedge mustDirtyAft[n](array) = True)$
// end_chkpt(array) will be placed along the edge $m \to n$ if
$placeEndChkptEdge(m, n, array) =$
    // node $n$ is itself clean but predecessor $m$ is dirty, $n$ contains or is followed
    // by a write and predecessor $m$ is not itself preceded by $end\_chkpt(array)$
    $(mustDirty[n](array) = False \wedge mayDirty[n](array) = True \wedge$
     $mayWillWrite[n](B) = True \wedge mustDirtyAft[m](array) = False \wedge$
     $mustEndChkptPlaced[m](array) = False)$
// start_chkpt(array) will be placed immediately after node $n$ if
$placeStartChkptNode(n, array) =$
    // node $n$ is the last write to $array$ until the next checkpoint
    $(mayWillWriteBef[n](array) = True\ \wedge\ mayWillWrite[n](array) = False)$
// start_chkpt(array) will be placed along the edge $m \to n$ if
$placeStartChkptEdge(m, n, array) =$
    // node $n$ follows the last write to $array$ until the next checkpoint
    $(mayWillWriteBef[n](array) = False \wedge mayWillWrite[m](array) = True)$
```

Fig. 6. Dataflow formulas for the loop-sensitive extension

While the `start_chkpt` rule it simple, it does not work unless the runtime knows the number of threads that will call `start_chkpt` between a pair of checkpoints. Unless this is known, it is not possible to determine whether a given thread's `start_chkpt` is the last one. OpenMP applications spawn new threads by executing code that is marked by `#pragma omp parallel`. Variables declared inside this block are private to each thread and variables declared outside this block may be either private or shared among the spawned threads, with heap data always being shared. A spawned thread may spawn additional threads of its own, making any variable it has access to either shared among all the newly spawned threads or private to each of them. We track the number of threads that a given array is shared among by adding calls to the `start_thread` and `end_thread` functions at the top and bottom of each `#pragma omp parallel` region, respectively:

- `start_thread(parentThread, privateArrays)` - Informs the checkpointing runtime that `parentThread` has spawned a new thread, giving it private copies of the arrays in the `privateArrays` list. All other arrays are assumed to be shared between `parentThread` and the new thread.
- `end_thread()` - Informs the runtime that the given thread is about to terminate.

Note that since making a shared variable private to a thread is equivalent to creating a copy of the original variable, the `start_thread` and `end_thread` calls imply `add_array` and `remove_array` calls, respectively, for the privatized variables.

Original Code	Code with Must-EndChkptPlaced States	Transformed Code
`...`	`... [A→F,B→F]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`end_chkpt(A);`
`...`	`[A→T,B→F]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→T]`	`start_chkpt(A);`
`  ...`	`  ... [A→T,B→T]`	`...`
`  B[...]=...;`	`  B[...]=...; [A→T,B→T]`	`end_chkpt(B);`
`  ...`	`  ... [A→T,B→T]`	`for(...) {`
`}`	`} [A→T,B→T]`	`  ...`
`...`	`... [A→T,B→T]`	`  B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→T]`	`  ...`
		`}`
		`start_chkpt(B);`
		`...`
		`checkpoint();`

Fig. 8. Transformation example with loop-sensitive optimizations

While these runtime monitoring mechanisms can identify the number of threads that *may* call `start_chkpt(array)`, it is not clear that all of these threads will actually perform the call, since different threads may execute different code. Appendix VI presents a proof (`Theorem 2`) that if one thread calls `start_chkpt(array)` between two checkpoints, all threads will do the same. Since it is possible for a thread to spawn new threads after calling `start_chkpt(array)`, the runtime considers such threads as having also called `start_chkpt(array)`, until the next `checkpoint` call. Similarly, if threads in a thread group call `start_chkpt(array)` before exiting their `#pragma omp parallel` region, the checkpointing runtime decrements the count of threads that have already called `start_chkpt(array)` appropriately. `Theorem 1` guarantees that if one such thread calls `start_chkpt(array)`, they all do.

V. Experimental Evaluation

A. Experimental Setup

We have evaluated the effectiveness of the above compiler analysis by implementing it on top of the ROSE [11] source-to-source compiler framework and applying it to the OpenMP versions [1] of the NAS Parallel Benchmarks [2]. We have focused on the codes `BT`, `CG`, `EP`, `FT`, `LU`, `SP`. `MG` was omitted from our analysis since it uses dynamic multi-dimensional arrays (arrays of pointers to lower-dimensional arrays), which requires additional complex pointer analyses to identify arrays in the code. In contrast, the other codes use contiguous arrays, which require no additional reasoning power. Each NAS code was augmented with a `checkpoint` call at the top of its main compute loop and one immediately after the loop.

The target applications were executed on all problem classes (`S`, `W A`, `B` and `C`, where `S` is the smallest and `C` the largest), on 4-way 2.4Ghz dual-core Opteron SMPs, with 16GB of RAM per node (Atlas cluster at the Lawrence Livermore National Laboratory). Each run was performed on a dedicated node, regardless of how many of the node's cores were actually used by the NAS benchmark. All results are averages of 10 runs and each application was set to checkpoint 5 times, with the checkpoints spaced evenly throughout the application's execution. This number was chosen to allow us to sample the different checkpoint sizes that may exist in different parts of the application without forcing the application to take a checkpoint during every single iteration, which would have been unrealistically frequent. Data for `BT`, `EP` and `FT` is not available at input size `C` because in these codes it requires too much static memory to compile.

The transformed codes were evaluated with two checkpointers. First, we used the Lazarus sequential incremental checkpointer, which implements page-protection-based incremental and asynchronous checkpointing. We extended this checkpointer with the API from Section II and used it to compare the performance of purely runtime and compiler-assisted incremental and asynchronous checkpointing. Since we did not have a multi-threaded checkpointer available to us, we developed a model checkpointing runtime to evaluate the performance of our compiler technique on multi-threaded applications. This model runtime implements the API from Section II and simulates the behavior of a real checkpointer. It performs the same state tracking and synchronization as a real checkpointer but instead of actually saving application state, it simply sleeps for an appropriate period of time. One side-effect is that our checkpointer does not simulate the overheads due to saving variables other than arrays. However, since in the NAS benchmarks such variables make up a tiny fraction of overall state, the resulting measurement error is small. Furthermore, because the model checkpointer can sleep for any amount of time, it can simulate checkpointing performance for a wide variety of storage I/O bandwidths.

The checkpointers have two major modes of operation: `PR` mode, where incremental checkpointing is done using a "Purely-Runtime mechanism", and `CA` mode, where "Compiler-Assistance" is used. Lazarus supports both modes, and the model checkpointer only supports the `CA` mode. Both checkpointers also support all four permutations of the following modes:
Checkpoint contents:

- `Full`: Lazarus saves the application's entire state and the model checkpointer simulates the time it would take to checkpoint all currently live arrays.
- `Incr`: Incremental checkpointing, where only the data that has been written to since the last checkpoint is saved. In `PR` mode the operating system's page protection mechanism is used to track whether a given page has been written to since the last checkpoint. In `CA` mode the checkpointer saves all arrays for which `start_chkpt` has been called since the last `checkpoint`.

Checkpoint timing:

- `Block`: Blocks the `checkpoint` call while saving the appropriate portion of application state.
- `Asynch`: Checkpointing is performed asynchronously. The model checkpointer uses a single extra thread. In `PR` mode Lazarus launches an extra process, using the `fork` system call. In `CA` mode it spawns a separate checkpointing thread for each application array.

B. Pure-Runtime vs. Compiler-Assisted Checkpointing

This section compares the performance of compiler-assisted checkpointing with purely-runtime checkpointing, using Lazarus.

1) Checkpoint Size: We begin begin by comparing the sizes of the checkpoints generated by using Lazarus in `PR` and `CA` modes. In the context of the NAS codes, which have an initialization phase, followed by a main compute loop, the primary effect of the analysis is to eliminate write-once arrays from checkpointing. These are the arrays that are written to during the initialization phase and then only read from during the main compute loop. As such, since there do not exist any `start_chkpt` calls for these arrays during the main compute loop, they are only saved during the first checkpoint and omitted in subsequent checkpoints. In contrast, Lazarus' page-protection-based mechanism can track any changes to application state, at a page granularity.

Figure 9 shows the % reduction in the sizes of checkpoints created by the two modes (data for `IS` is not available due to technical issues with Lazarus). For each application we show results for `PR` and `CA` modes for all input sizes. It can be seen that while for small inputs there is sometimes a difference between the effectiveness of the two approaches, for larger input sizes the difference becomes very small. This is because these benchmarks overwrite enough of their state that a page-granularity-based detection method cannot identify regions of memory that have not changed from checkpoint to checkpoint. This shows that it is possible for a compiler-based mechanism to achieve the same reductions in checkpoint size as are available from a runtime-based technique, without any runtime monitoring. Since both the compiler technique presented in this paper and page-based incremental checkpointing operate at a fairly coarse grain of whole arrays and whole pages, in future work we plan to compare finer grain runtime techniques [4] to more accurate compiler-based techniques.

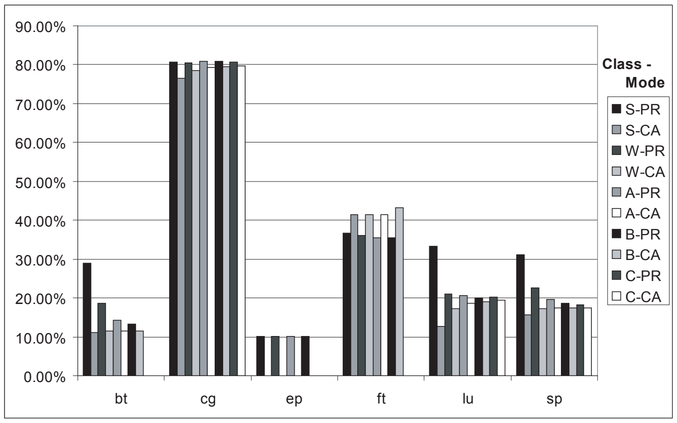


Fig. 9. Checkpoint Sizes Generated by Lazarus in `PR` and `CA` modes

2) Running Time: The impact of the various checkpointing mechanisms on the application running time was evaluated by looking at (i) the cost of using the given mechanism, without actually writing anything to disk and (ii) the cost of writing to disk using the mechanism. Figure 10 shows the running time of each NAS application on input size `A` (other inputs show similar results), both without Lazarus (`Original` column) and with each Lazarus checkpointing configuration where no checkpoint data is written to disk. It can be seen that the checkpointing mechanisms themselves add very little cost to original application and all checkpointing mechanisms have similar base overheads.

Figure 11 shows the difference between the above times and the running times of these configurations on input size `A`, where Lazarus writes checkpoints to the parallel file system. This is the time spent writing checkpoints under each scheme. It can be seen that pure-runtime incremental checkpointing has the smallest ultimate impact on application performance by a wide margin, with the asynchronous algorithm being generally faster than the synchronous algorithm. Incremental checkpointing is generally much cheaper than recording full checkpoints but asynchronous checkpointing is not always faster than blocking checkpointing because of the additional synchronization costs. Compiler-assisted checkpointing is more expensive than pure-runtime checkpointing. However, it is not clear how much of this is due to the basic approach and how much to the implementation details between the two checkpointing runtimes (fork vs. one checkpointing thread per array). We expect to examine this in the future by augmenting existing multi-threaded checkpointers such as BLCR [7] with the API from Section II.

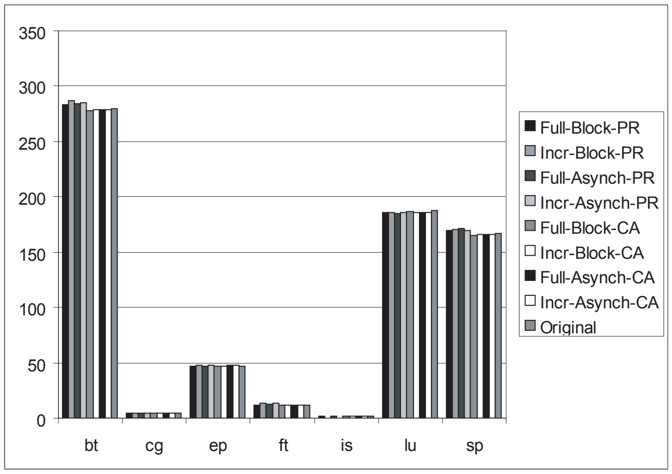


Fig. 10. Execution times with all Lazarus configurations, no data written (input size `A`)

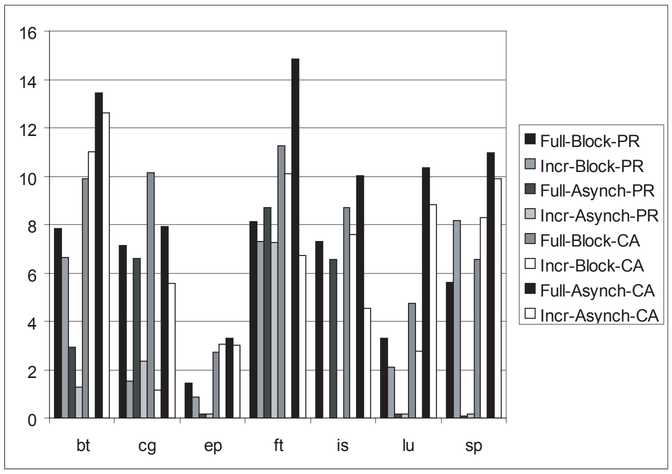


Fig. 11. Checkpointing time for all Lazarus configurations, data written to parallel file system (input size `W`)

C. Compiler-Assisted OpenMP Checkpointing

This section looks at the cost of checkpointing for multi-threaded OpenMP applications, using the model checkpointing runtime.

1) Checkpoint Size: Figure 12 shows the % reduction in the sizes of checkpoints as a result of using the compiler analysis for input size `A` (very similar numbers for other input sizes). The amount of savings varies between different applications, ranging from 79% for `CG` to 0% for `EP` with 1 thread. There is little change in the reduction as the number of threads increases. The only exceptions are `EP` where it ranges from 0% for 1 thread to 17% for 8 threads and `IS`, which ranges from 15% for 1 thread to 26% for threads.

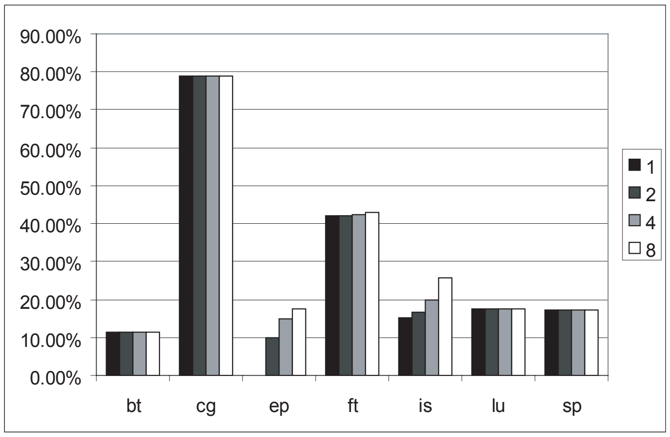
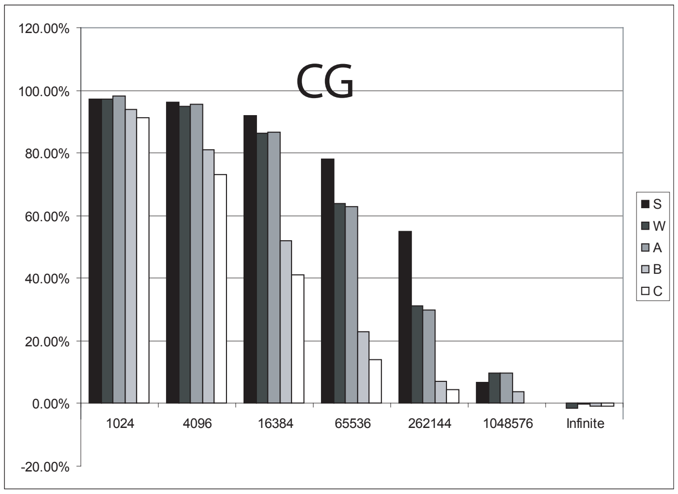


Fig. 12. Checkpoint sizes generated by the model checkpointer (input size `A`)

2) Running Time - Incremental Checkpointing: Figure 13 shows the the % reduction in the running times of the NAS applications, running with the model checkpointer in configuration `Incr-Block` relative to using configuration `Full-Block`. We show only `CG` and `EP` on 4 threads, since the behavior of these codes is similar to that of other codes that have a small or a large checkpoint size reduction, respectively, and the number of threads has no effect on this behavior. The x-axis is the I/O bandwidth used in the experiments, ranging from 1 MB/s to 1 GB/s in multiples of 4, including a datapoint for infinite bandwidth. This range includes a variety of use-cases, including hard-drives (60MB/s write bandwidth) and 10 Gigabit Ethernet(1GB/s bandwidth). For `EP`, although there is some difference in performance between the two configurations, the effect is generally small and always $< 20\%$ at all bandwidths. However, for `CG`, the effect is quite dramatic, with the improvement from `IncrChkpt-Block` ranging from 95% for low bandwidths, when the cost of checkpointing is important, to $< 10\%$ high bandwidths.

D. Running Time - Asynchronous Checkpointing

We evaluated the performance of asynchronous checkpointing by comparing the execution time of applications that use `Incr-Asynch` to those that use



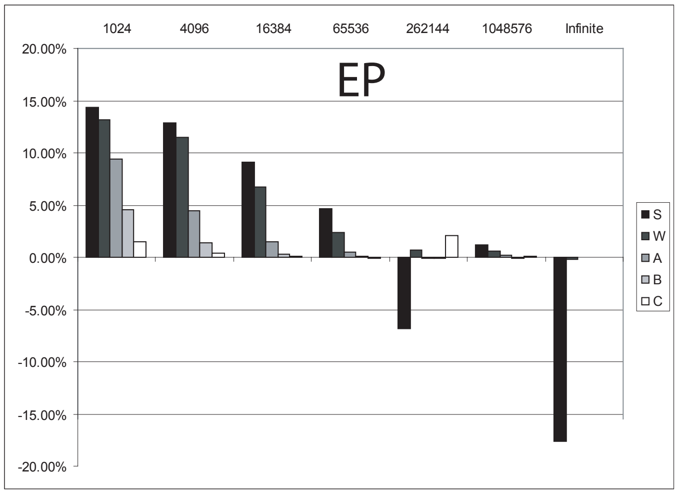


Fig. 13. Execution time differences between `Incr-Block` and `Full-Block`

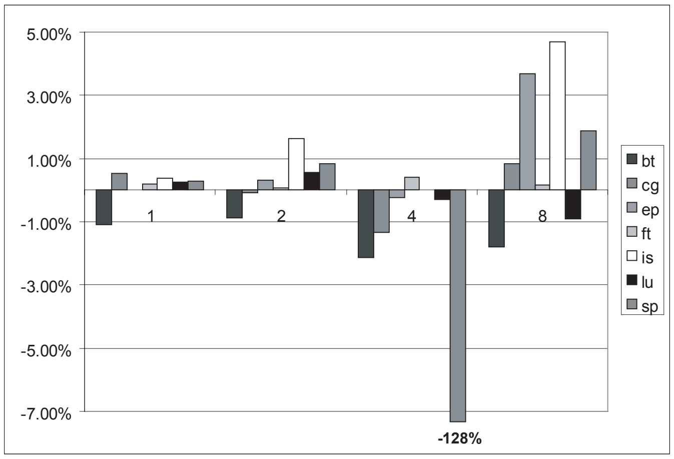


Fig. 14. Relative execution time differences between `Incr-Asynch` and `Incr-Block` with infinite bandwidth (input size `A`)

`Incr-Block`. Figure 14 shows the % reduction in application running times from using with `Incr-Asynch` instead of `Incr-Block` for input size `A` (same patterns for other input sizes). No data was saved to highlight the raw overhead of the runtime mechanisms required for blocking and asynchronous checkpointing. For most applications there is very little difference between the two configurations. The only exception is `SP` on 4 threads, where running with `Incr-Asynch` more than 2x slower than running with `Incr-Block`. The reason appears to be the addition of the extra thread. The nodes we were using have different memory banks associated with different pairs of processors. As such, an the extra checkpointing thread that is assigned to a processor that does not share a memory bank with the main computing processors, will suffer from poor synchronization performance. In contrast, blocking checkpointing has no extra thread and requires no additional synchronization.

Figure 15 shows the same configurations but with the full range of bandwidths. Data for 2-thread and 4-thread runs is presented, since it is typical. Asynchronous checkpointing tends to perform better than blocking checkpointing for most bandwidths and applications, although the improvement does not hold in all cases. `CG` performs worse with asynchronous checkpointing for small bandwidths for all thread numbers and `SP` shows a large slowdown with asynchronous checkpointing.

VI. Summary

We have presented a novel compiler analysis for optimizing automated checkpointing. Given an application that has been augmented by the user with calls to a `checkpoint` function, the analysis identifies regions in the code that do not have any writes to each given array. This information can be used to reduce the amount of data checkpointed and to asynchronously checkpoint this data in a separate thread. In our experiments with the NAS Parallel Benchmarks we have found that this analysis can reduce checkpoint sizes by as much as 95%. These checkpoint size reductions were found to have a notable effect on checkpointing performance. Furthermore, we evaluated the performance of compiler-enabled asynchronous checkpointing. Although our experiments showed that asynchronous checkpointing is frequently better than blocking checkpointing, we discovered that this is oftentimes not the case, meaning that the choice of the optimal checkpointing technique closely depends on the application. These results also suggest that more work should be done to understand of the performance characteristics of asynchronous checkpointing runtime systems.

This work performed under the auspices of the U.S. Dept. of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

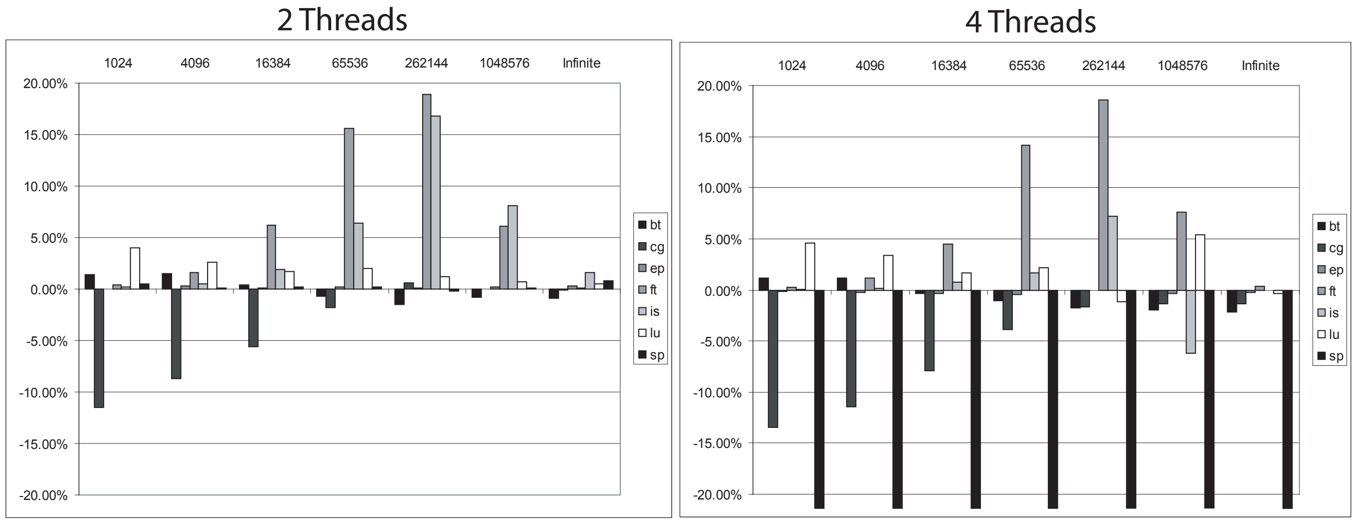


Fig. 15. Execution time differences between `Incr-Asynch` and `Incr-Block` (input size `A`)

References

[1] http://phase.hpcc.jp/Omni/benchmarks/NPB.

[2] http://www.nas.nasa.gov/Software/NPB.

[3] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, and J. An overview of the BlueGene/L supercomputer. In *IEEE/ACM Supercomputing Conference*, 2002.

[4] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th International Conference on Supercomputing (ICS)*, pages 277 – 286, 2004.

[5] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level checkpointing for shared memory programs. October 2004.

[6] Greg Bronevetsky, Paul Stodghill, and Keshav Pingali. Application-level checkpointing for openmp programs. In *International Conference on Supercomputing*, June 2006.

[7] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2006.

[8] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, November 2005.

[9] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratorys ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[10] James S. Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.

[11] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.

[12] Kim Cupps Rob Ross, Jose Moreirra and Wayne Preiffer. Parallel I/O on the IBM Blue Gene/L system. Technical report, BlueGene Consortium, 2005.

[13] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 58, 2004.

[14] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[15] Kun Zhang and Santosh Pande. Efficient application migration under compiler guidance. In *Poceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 10–20, 2005.

Appendix: OpenMP Theorems

`Lemma 1`: Let $n_1 \rightarrow^* n_k$ be a path in the CFG from node n_1 to node n_k such that $mayWillWriteBef[n_k](array) = False$ and no node $n_2, ..., n_{k-1}$ contains a `checkpoint` call. Then, $\forall 1 < l < k.$

$$mayWillWriteBef[n_l](array) = True \Leftrightarrow$$
$$\forall 1 < p < l.\ mayWillWriteBef[n_p](array) = True$$

`Proof`:

`Case` $\Rightarrow$:

- Suppose that $mayWillWriteBef[n_l](array) = True$ and $\exists 1 < p < l.$ $mayWillWriteBef[n_p](array) = False.$
- Let p' be the largest such number. As such, $mayWillWriteBef[n_{p'}](array) = False,$ $mayWillWriteBef[n_{p'+1}](array) = True.$
- $mayWillWrite[n_{p'}](array) = \bigcup_{m \in succ(n_{p'})} mayWillWriteBef[m](array).$
- Since $mayWillWriteBef[n_{p'+1}](array) = True$, we know that $mayWillWrite[n_{p'}](array) = True.$
- $mayWillWriteBef[n_{p'}](array) = [\![n_{p'}]\!](mayWillWrite[n_{p'}](array), array).$
- However, the only statement that can cause a $True$ to $False$ transition is a `checkpoint` call and we have assumed that this cannot happen.

`Case` $\Leftarrow$:

- Assume that $\forall.\ 1 < p < l.$
  $mayWillWriteBef[n_p](array) = True.$
- This means that
  $mayWillWriteBef[n_{l-1}](array) = True.$
- $mayWillWrite[n_l](array) =$
  $\bigcup_{m \in succ(n_l)} mayWillWriteBef[m](array).$
- As such, $mayWillWrite[n_l](array) = True.$
- $mayWillWriteBef[n_l](array) =$
  $[\![n_l]\!](mayWillWrite[n_l](array), array).$
- Since node n_l can't be a `checkpoint` call, we know $mayWillWriteBef[n_l](array) = True.$

`Lemma 2`: Let $n_1 \rightarrow^* n_k$ be a path as above.
$\forall 1 < l < k.$
$mayWillWriteBef[n_l](array) = False \Rightarrow$
$\forall l \leq p \leq k.$
$mayWillWriteBef[n_p](array) = False.$
`Proof`:

- Assume that
  $mayWillWriteBef[n_l](array) = False.$
- Suppose that $\exists l \leq p \leq k.$
  $mayWillWriteBef[n_p](array) = True$
- If $l = p$ we have a contradiction, since
  $mayWillWriteBef[n_l](array) = False.$
- Otherwise, $l < p$ and `Lemma 1` imply that
  $mayWillWriteBef[n_l](array) = True.$
- This contradicts our assumption.

`Lemma 3`: Let $n_1 \rightarrow^* n_k$ be a path as above.
$\exists 1 \leq l < k.$ `start_chkpt(array)` appears immediately after node n_l or on the edge $n_l \rightarrow n_{l+1} \Rightarrow$
$\forall 1 \leq i < l.\ mayWillWrite[n_i](array) = True$ and
$\forall l \leq j \leq k.\ mayWillWriteBef[n_j](array) = False.$
`Proof`:

- Since $\exists$ a `start_chkpt(array)` call along the path, we know that either
  (i) $mayWillWriteBef[n_l](array) = True\ \wedge$
  $mayWillWrite[n_l](array) = False$ or
  (ii) $mayWillWrite[n_l](array) = True\ \wedge$
  $mayWillWriteBef[n_{l+1}](array) = False.$
- $i:\ 1 \leq i < l$:
  - If (ii) is true, we know that $mayWillWriteBef[n_l](array) = True$ because $mayWillWriteBef[n_l]](array) =$ $[\![m]\!](mayWillWrite[n_l]](array), array)$ and the node n_l is not a `checkpoint` call by assumption.
  - As such, in both cases:
    $mayWillWriteBef[n_l](array) = True.$
  - From `Lemma 1` we know that
    $\forall 1 < p \leq l.$
    $mayWillWriteBef[n_p](array) = True.$
  - $mayWillWrite[n](array) =$
    $\bigcup_{m \in succ(n)} mayWillWriteBef[m](array).$
  - As such, $\forall 1 \leq i < l.$
    $mayWillWrite[n_i](array) = True.$
- $j:\ l \leq j < k$
  - If (i) is true, we know that $mayWillWriteBef[n_{l+1}](array) = False$ because $mayWillWrite[n_{l+1}](array) =$ $\bigcup_{m \in succ(n_{l+1})} mayWillWriteBef[m](array).$
  - As such, in both cases:
    $mayWillWriteBef[n_{l+1}](array) = False.$
  - According to `Lemma 2`, $\forall l \leq p \leq k.$
    $mayWillWriteBef[n_p](array) = False$

`Theorem 1`: Let $n_1 \rightarrow^* n_k$ and $n'_1 \rightarrow^* n'_k$ be two paths as above, with $n_1 = n'_1$ and $n_k = n'_k$.
If $\exists$ a `start_chkpt(array)` call on one path then $\exists$ a `start_chkpt(array)` call on the other path.
`Proof`:

- Assume that $\exists$ a `start_chkpt(array)` call along path $n_1 \rightarrow n_2 \rightarrow^* n_{k-1} \rightarrow n_k$.
- From `Lemma 3` we know that
  $\forall 1 \leq i < l.\ mayWillWrite[n_i](array) = True$ and $\forall l \leq j \leq k.$
  $mayWillWriteBef[n_j](array) = False.$
- As such, $mayWillWrite[n_1](array) = True$ and $mayWillWriteBef[n_k](array) = False.$
- Let l be the largest number s.t. $1 < l < k$ and $mayWillWrite[n_l](array) = True.$
- Thus, either (i) $l = k$ or
  (ii) $mayWillWrite[n_{l+1}](array) = False.$
- Suppose (i):
  - By the `start_chkpt` placement rules of Figure 3, `start_chkpt(array)` would be placed on edge $n_l \rightarrow n_k$.
- Now suppose (ii):
  - Either $mayWillWriteBef[n_{l+1}](array) = True$ or $= False.$
  - If $= True$, the the `start_chkpt` placement rules of Figure 3 would place `start_chkpt(array)` immediately after n_{l+1}.
  - If $= False$, the rules place place `start_chkpt(array)` on edge $n_l \rightarrow n_{l+1}$.

`Theorem 2`: Let n_1 be the start of the application or a `checkpoint` call and let n_k be a `checkpoint` call or the end of the application. For any two paths through the CFG from n_1 to n_k that do not contain `checkpoint` calls, if one contains a call to `start_chkpt(array)`, so does the other.
`Proof`: Direct application of `Theorem 1`.